DHRUV SINGAL   dsingal   349-01

Brute Force

Data Structures:

int[] values int[] weights, ArrayList<String> binStr: to store 2^n strings of n length returned by getGrayCode , int[] binInt: to store the integer array representation of a string in binStr, int[] maxProfitItems: to keep track of the items picked up to find the max feasible value

Algorithm:

The algorithm goes over each string returned from getGrayCode, turns it into an integer array, and multiplies that vector with the vector represented by the values array and the vector represented by the weights array to find the value and weight of the knapsack when items are picked up according to the string. If the value is greater than a previously seen value and the the total weight is less than the capacity, the maximum weight, value, and maximum string are updated to reflect this new knapsack configuration.

Optimization:

Currently, the algorithm is does 3n operations for each string. Therefore, the complexity is around $3n*2^n$. The factor 3n can brought down to 2n by having getGrayCode return an ArrayList of Integer arrays, so that the algorithm doesn't need to go through the string to convert it into integers for the dot product with the values and weights vector.

Pros: Guaranteed optimal knapsack configuration.

Cons: Performance degrades exponentially as n increases, so not feasible for large n both in terms of memory required and amount of computation required to arrive at the optimal knapsack configuration.

Greedy Search

Data Structures:

int[] values, int[] weights, int[] maxProfitItems: to keep track of the items picked up to find the max feasible value, double[] vWRatio: keeps track of the value/weight ratio of the n items, int[] identifiers, ArrayList<Integer> sortedIdentifiers: identifiers sorted in descending order of the value/weight ratio of the items they refer to.

Algorithm:

The identifiers are sorted in the descending order of the value/weight ratio of the items they refer to(nlogn). Then we go through the items in descending value/weight order and try to put them into the Knapsack, stopping once the Knapsack is full or when we've been through all the items(n). Every time we put something into the Knapsack, we increase it's value and decrement it's remaining capacity and set the item to have been picked in maxProfitItems[].

Optimization:

Currently, the algorithm requires nlogn comparisons to sort the identifiers based on the value/weight of the item they refer to, and then n comparisons to go through the list of items sorted in descending order. Thus the asymptotic complexity is O(nlogn). If we were to be

provided the items and identifiers sorted by their value/weight, we could avoid the sort, and thus only need n comparisons for the traversal, leading to an asymptotic complexity of O(n). Furthermore, if the values and weights of the items were dsuch that the ratio of the value to the weight was always a whole number, we could have the vWRatio array be all integers, thus needing less memory to store them, and less computation to compare, since comparing integers is easier than comparing floating point numbers.
Pros: Fast, and requires much less memory.
Cons: Doesn't provide the optimal Knapsack configuration for the 0/1 Knapsack problem, as putting one of the higher value/weight items in the Knapsack can cause there not to be enough space for a lower value/weight item that could have been fit otherwise and lead to wasted capacity in the Knapsack.

Dynamic Programming:
For dynamic programming, we fill in the table using the recurrence relation:

maxVal(i,w)=max(value(i)+maxVal(i-1,w-weight(i)), maxVal(i-1,w))

where w is the capacity of the knapsack, weight(i) is the weight of the ith item, value(i) is the value of the ith item, and maxVal(i,w) is the maximum value that can be obtained looking at i items and having a knapsack of capacity w. For the max(), we do value(i)+maxVal(i-1,w-weight(i)) only if the ith item would fit in the Knapsack.
Data Structures:
int[] values, int[] weights, int[] maxProfitItems: to keep track of the items picked up to find the max feasible value, int[][] table: dimensions = (n+1)X(capacity+1) to keep track of the maximum feasible value for a given number of items and knapsack capacity.
Algorithm:
The table is initialized with zeros in the first row and first column, corresponding to 0 items and 0 knapsack capacity respectively. We loop over each each row from left to right, using the recurrence relation to fill up the table. maxVal(i-1,w-weight(i)) is the value of table[i-1][j-weight(i)], where i iterates over the rows and j iterates over the columns.
After the table is filled, the maximum value of the Knapsack can be found at table[n][capacity]. Now, knowing the value, we have to backtrace through the table to find the weight of this optimal Knapsack configuration.
To backtrace, we try to find where the value in table[i][j] came from. We do this until table[i][j] is equal to 0. When we find an item we picked, we add the weight of that item, ie weights[i] to the maximum weight and set maxProfitItems[i] to 1 to represent having picked up the item.
Optimization:
There isn't much that can be done to make the dynamic approach better. If there wasn't a requirement to print out the items in sorted order, we could have gotten rid of the maxProfitItems[] and just printed out the items in descending order whilst we did the traceback. So the output would be item 20, 19, 18, 16, …… 3, 1 instead of 1, 3, ….., 16, 18, 19, 20.

Furthermore, since we test the weights and item compatibility during the filling of the table, we wouldn't have to do the traceback if we were interested only in the maximum value attainable, not the items or the weight of the final knapsack.
Pros: Fast, and requires less memory than Brute force and gives us a feasible Knapsack configuration with the most value attainable.
Cons: Uses more memory than Greedy Search and is slower than Greedy. (but gives globally optimal solution).

Branch and Bound:
Branch and Bound uses a fractional upper bound since it's tighter, and thus can prune faster at the cost of computational overhead whilst calculating the upper bound.
Data Structures:
int[] values: values of the n items, int[] weights: weights of the n items, int[] maxProfitItems: to keep track of the items picked up to find the max feasible value, double[] vWRatio: keeps track of the value/weight ratio of the n items, int[] identifiers: identifiers for the n items, ArrayList<Integer> sortedIdentifiers: identifiers sorted in descending order of the value/weight ratio of the items they refer to., Node with attributes String solution, double ub, int value, int remainingCap, PriorityQueue<Node> pq: priority queue with Nodes sorted order of their upper bounds in decreasing order.
Algorithm:
The identifiers are sorted in the descending order of the value/weight ratio of the items they refer to. We add the first node to the queue pq that uses a comparator to keep highest upper bound node at the head. While the queue isn't empty and all the nodes in the queue don't have upper bounds lower than already found solution (this can be tested as the highest upper bound node in the queue is at the head), we pop a node from the queue and check if it the last parent in it's branch of the state space tree.
If it is, then we try to give it the last item (ie, the lowest value/weight) item, and change the maxValue and maxString(the solution string of the node) seen if the new value is greater than what we've seen before.
If the popped node isn't the last parent, then we create two new nodes, a left and a right. Left doesn't take the next item and it's upper bound is computed. It's capacity and remaining weight remain unchanged and it's enqueued.
We create the right node only if the node has enough remaining capacity to take the next highest value/weight item. It takes the item and it's upper bound is computed and it's enqueued.
How upper bound is computed: for a node, we compute it's upper bound by greedily going over the highest value/weight ratio item that it hasn't seen and add their value to the value of the parent of the node + (value of the item if it's right) if it can take these items whole. If it can't take these items whole, we split the item, and increase the upper bound by (remaining capacity of the node) * (value/weight of the item).

Optimization:

We can reduce the number of comparisons that need to be done if the items are provided in already sorted order by value/weight.

Pros: Fast, almost twice as fast as Dynamic Programming under certain favourable values that allow for fast pruning.

Cons: Can be led awry if the values are such that don't allow for fast pruning and don't let us have tight bounds. Can crash if n is too large and we can't prune nodes, leading to too many nodes that need to be kept in memory.

Supply Table

	Easy20	Easy50	Hard50	Easy200	Hard200
Enum	726(519) 852 ms	N/A	N/A	N/A	N/A
Greedy	692(476) 76 ms	1115(247) 73 ms	16538(10038) 68 ms	4090(2655) 97 ms	136724(111924) 96 ms
Dyn Prog	726(519) 4 ms	1123(250) 6 ms	16610(10110) 24 ms	4092(2658) 30 ms	137448(112648) 167 ms
B'n'B	726(519) 4 ms	1123(250) 7 ms	16610(10110) 7 ms	4092(2656) 24 ms	136967(112167) 60308 ms (run only for 1 minute)